

# TFE4171 - Project Part A

Jens Fredrik Bergem & Toushif Islam Shoueb

March 2025

# 1 Immediate assertions

All assertion have used the ReadAddress() function to pass in the address for the RxSC and Rx buffer register, and to retrieve the corresponding data stored.

## 1.1 VerifyAbortReceive

The assertions in the VerifyAbortReceive task checks that Rx\_Ready is 0, Rx\_FrameError is 0, Rx\_AbortSignal is 1, and that Rx\_Overflow is 0 in the RxSC register. There's also an assertion making sure that data in the Rx buffer is zero after abort has been asserted.

```
35  task VerifyAbortReceive(logic [127:0][7:0] data, int Size);
36    // INSERT CODE HERE
37    logic [7:0] ReadData;
38    ReadAddress(2, ReadData);
39    assert (!ReadData [0]) else $display("VerifyAbortReceive FAIL : Rx_Ready bit is not low");
40    assert (!ReadData [2]) else $display("VerifyAbortReceive FAIL : Rx_FrameError bit is not low");
41    assert (!ReadData [3]) else $display("VerifyAbortReceive FAIL : Rx_AbortSignal bit is not high");
42    assert (!ReadData [4]) else $display("VerifyAbortReceive FAIL : Rx_Overflow bit is not low");
43
44    ReadAddress(3, ReadData);
45    assert (ReadData == 0) else $display("VerifyAbortReceive FAIL : Rx data buffer is not zero");
46  endtask
```

Figure 1: Task VerifyAbortReceive

## 1.2 VerifyNormalReceive

The assertions in the VerifyNormalReceive task checks that Rx\_Ready is 1, Rx\_FrameError is 0, Rx\_AbortSignal is 0, and that Rx\_Overflow is 0 in the RxSC register. We're also looping through each row in the data matrix, and making sure that the current data read from the ReadAddress() function corresponds to the data stored in the current row of the matrix.

```
50  task VerifyNormalReceive(logic [127:0][7:0] data, int Size);
51    logic [127:0][7:0] ReadData;
52    wait(uin_hdlc.Rx_Ready);
53
54    // INSERT CODE HERE
55    ReadAddress(2, ReadData);
56    assert (ReadData [0]) else $display("VerifyNormalReceive FAIL : Rx_Ready bit is not high");
57    assert (!ReadData [2]) else $display("VerifyNormalReceive FAIL : Rx_FrameError bit is not low");
58    assert (!ReadData [3]) else $display("VerifyNormalReceive FAIL : Rx_AbortSignal bit is not low");
59    assert (!ReadData [4]) else $display("VerifyNormalReceive FAIL : Rx_Overflow bit is not 34243241low");
60
61    for (int i = 0; i < Size; i++) begin
62      ReadAddress(3, ReadData[i]);
63      assert (ReadData[i] == data[i]) else $display("VerifyNormalReceive FAIL : Rx data buffer does not contain the correct value");
64    end
65  endtask
```

Figure 2: Task VerifyNormalReceive

### 1.3 VerifyOverflowReceive

The assertions in the VerifyAbortReceive task checks that Rx\_Ready is 1, Rx\_FrameError is 0, Rx\_AbortSignal is 0, and that Rx\_Overflow is 1 in the RxSC register. Again, we also have assertions for checking that the data buffer contains the correct data.

```
69  task VerifyOverflowReceive(logic [127:0][7:0] data, int Size);
70    logic [127:0][7:0] ReadData;
71    wait(uin_hd1c.Rx_Ready);
72
73    // INSERT CODE HERE
74    ReadAddress(3'b010, ReadData);
75    assert (!ReadData[0]) else $display("VerifyOverflowReceive FAIL : Rx_Ready bit is not high");
76    assert (!ReadData[2]) else $display("VerifyOverflowReceive FAIL : Rx_FrameError bit is not low");
77    assert (!ReadData[3]) else $display("VerifyOverflowReceive FAIL : Rx_AbortSignal bit is not low");
78    assert (!ReadData[4]) else $display("VerifyOverflowReceive FAIL : Rx_Overflow bit is not high");
79
80    for (int i = 0; i < Size; i++) begin
81      ReadAddress(3, ReadData[i]);
82      assert (ReadData[i] == data[i]) else $display("VerifyOverflowReceive FAIL : Rx data buffer does not contain the correct value");
83    end
84  endtask
```

Figure 3: Task VerifyOverflowReceive

## 2 Concurrent assertions

### 2.1 Rx\_Flag detection

The sequence for the Rx\_flag was 01111110 according to the specification, so we defined a sequence that resembles this, before adding it to the property RX\_FlagDetect.

```
38  sequence Rx_flag;
39    // INSERT CODE HERE
40    !Rx ##1 Rx [*6] ##1 !Rx;
41  endsequence
42
43  // Check if flag sequence is detected
44  property RX_FlagDetect;
45    @(posedge Clk) Rx_flag |-> ##2 Rx_FlagDetect;
46  endproperty
47
48  RX_FlagDetect.Assert : assert property (RX_FlagDetect) begin
49    $display("PASS: Flag detect");
50  end else begin
51    $error("Flag sequence did not generate FlagDetect");
52    ErrCntAssertions++;
53  end
```

Figure 4: Property RX\_FlagDetect

## 2.2 Rx\_Abort detection

The sequence for the Rx\_Abort was 11111110 according to the spec, so we defined a sequence that resembles this, before adding it to the property RX\_AbortSignal.

```
59 | sequence Rx_Abortdetect;
60 |   Rx [*7] ##1 !Rx;
61 | endsequence
62 |
63 | //If abort is detected during valid frame, then abort signal should go high
64 | property RX_AbortSignal;
65 |   // INSERT CODE HERE
66 |   @(posedge Clk) Rx_Abortdetect ##0 Rx_ValidFrame |=> Rx_AbortSignal;
67 | endproperty
68 |
69 | RX_AbortSignal.Assert : assert property (RX_AbortSignal) begin
70 |   $display("PASS: Abort signal");
71 | end else begin
72 |   $error("AbortSignal did not go high after AbortDetect during validframe");
73 |   ErrCntAssertions++;
74 | end
```

Figure 5: Property RX\_AbortSignal

## 3 Immediate vs Concurrent assertions

The main difference between immediate and concurrent assertions are that immediate assertions are evaluated at the current simulation time whereas concurrent assertions are evaluated over multiple units of time/clock cycles. In this case for our immediate assertions, we are evaluating whether or not the buffer and status control register contains the correct values at the current time and we are not concerned with the prior sequence of inputs. In our concurrent assertions we are evaluating whether or not a flag/abort sequence is detected, and because there's a sequence involved we need to evaluate the signals over multiple clock cycle instead of using immediate assertions.

## 4 RXSC issue

Upon simulating for the first time we encountered an issue with RXSC - it was not declared anywhere in the code. To fix this we simply passed in the value "2" instead, as the RXSC variable represents the RXSC register address which is 0x2.

```
239     //Enable FCS
240     if(!Overflow && !NonByteAligned)
241         WriteAddress(2, 8'h20); // RXSC replaced with 2
242     else
243         WriteAddress(2, 8'h00); // RXSC replaced with 2
244
```

Figure 6: Resolving RXSC issue

## 5 Result of simulation

Result of simulation showed that all assertions expect for the overflow bit in VerifyOverflowReceive passed. We are unsure why this one failed as our immediate assertion looks correct according to the specification.

```
# ****
#          0 - Starting Test Program
# ****
# ****
#          1000 - Starting task Receive - Normal
# ****
# PASS: Flag detect
# PASS: Flag detect
# ****
#          78250 - Starting task Receive - Abort
# ****
# PASS: Flag detect
# ****
#          268250 - Starting task Receive - Overflow
# ****
# PASS: Flag detect
# PASS: Flag detect
# VerifyOverflowReceive FAIL : Rx_Overflow bit is not high
# ****
#          948250 - Starting task Receive - Normal
# ****
# PASS: Flag detect
# PASS: Flag detect
# ****
#          1202750 - Starting task Receive - Normal
# ****
# PASS: Flag detect
# PASS: Flag detect
# ****
#          1866250 - Starting task Receive - Abort
# ****
# PASS: Flag detect
# ****
#          2392750 - Starting task Receive - Overflow
# ****
# PASS: Flag detect
# PASS: Flag detect
# VerifyOverflowReceive FAIL : Rx_Overflow bit is not high
# ****
#          3067750 - Starting task Receive - Normal
# ****
# PASS: Flag detect
# PASS: Flag detect
# ****
#          3223250 - Starting task Receive - Normal
# ****
# PASS: Flag detect
# PASS: Flag detect
# ****
#          3488250 - Finishing Test Program
# ****
```

Figure 7: Simulation result